

Collaboratory Workshop - Introduction to Python - Day 1/3

In this day we will discuss the format of the workshop, Spyder, IPython, and Python, assignment and variables, mathematical operations, reassignment, floats and integers, object types and coercion, text (strings), logical in/exclusion, indexing, true/false conditionals, if/else statements, functions, and modules.

This workshop aims to teach you some of the basics of coding in Python. Note that as the Collaboratory offers other workshops focusing on different aspects of Python (graphing and data analytics, digital image processing, etc.) we will not cover many things that you may otherwise find in a first 'course' in programming.

The workshop is designed in two parts for each day. In the first (and longest) part we are introduced to different aspects of Python. In this first portion you are reading instructions from the .py files (or from what the instructor says aloud) and typing code along with the instructor. In the second portion you will work on implementing the techniques introduced, but with slight variations on the goals and or methods. After each day has finished, I will e-mail copies of the tutorials with my own code written into the blanks. But, don't let that demotivate you from typing anything. Most of programming is troubleshooting, so I encourage you to type along, and we will address any questions and errors as they arise. My own coding results in errors every single day, it is a natural part of programming.

So, what is Spyder and how does it communicate with Python? The Spyder Integrated Development Environment (IDE) is a computer program that has both IPython and a text editor. You are reading from the text editor.

IPython is an interpreter for Python. It is a computer application that allows you to execute Python commands and programs.

Python itself is a programming language. It is the language that you write computer programs in.

You can think of Python as the grammar and style with which you write code, IPython then reads, interprets, and executes the code (hence being called an interpreter), and Spyder is software that allows to do all of these things at once in a structured environment.

Within Spyder, we have the text editor (the window in which you are currently reading text) the variable/file explorer and help menu on the upper right, and the IPython console in the bottom right.

We can execute code both in the console directly, or by running chunks of code (cells) from the text editor file, or even the entire text editor file itself. Spyder does not run code line by line.

'###' Defines a cell. With each cell we can write a block of code that can be executed independently of the other cells using hot keys (or by clicking on the right pointing green arrow button). This gives us a natural way to set up a tutorial, where every block of code that we want to run is entered into a separate cell. This may seem arduous at first (it is not how Python is intended to be run), but hopefully it will pay off by providing the tactile experience of programming. As we learn more we will transition to more standard uses of Python (scripting), and will also experiment with something called Jupyter (essentially a web-based version of Spyder).

Separately, a single pound symbol will define a 'comment', or 'commented code'. This is code that the interpreter will not read, and hence will not be executed, allowing you to narrate and describe what your code does. It is very important that you develop a healthy habit of ALWAYS COMMENTING YOUR CODE. We do this so that when sharing our code with others, they can quickly understand what the code does.

To 'execute' a cell in Spyder, press Ctrl+Enter. Try it in this cell. Nothing should happen because this cell is commented. To execute a cell and advance to the next cell, press Ctrl+Shift+Enter

Excellent. Now we will execute our first line of actual Python code. Execute this cell of code, which contains the following 'print()' function, and the argument 'hello world'.

```
print('hello world')
```

In the console you should see the input to 'print('hello world')', and the output 'hello world'.

```
### ----- ASSIGNMENT OF SIMPLE VARIABLES -----
```

```
# The basic variables that are common across many languages are strings, integers, floats, and booleans. Strings are collections of letters, numbers, and punctuation. Integers are the numerical integers (including positives and negatives). Floats are integers with decimal values. Booleans are the values 'True' or 'False'. The use of 'True' and 'False' in programming is fundamental. Other variable types do exist, and we will explore them, but they are particular to Python, where as those we have just mentioned are common to all interpreted languages (e.g. R and Matlab).
```

```
### Assign the string "this our 1st string" to the variable named "sometext"
```

```
sometext = "this is our 1st string"
```

```
### Now tell Python to print the value of the variable "sometext" using the print() function.
```

```
print(sometext)
```

```
### Assign the integer 42 to the variable name "someinteger"
```

```
someinteger = 42
```

```
### Now tell Python to print the value of the variable "someinteger"
```

```
print(someinteger)
```

```
### Now tell Python to print the values of both of the variables "sometext" and "someinteger".
```

Hint: this can be done in more than one way.

```
print(sometext, someinteger)
```

```
print(sometext)
```

```
print(someinteger)
```

Notice that if you type only the names of variables, and that is the final piece of code in a cell, or script, or console entry, then the value of the final variable in that group will be displayed.

```
sometext
```

```
someinteger
```

Now assign the value of 'True' to the variable name "aboolean"

```
aboolean = True
```

Now tell Python to print the value of "aboolean".

```
print(aboolean)
```

----- MATHEMATICAL OPERATIONS -----

Here we practice using different mathematical operations. All of your favorite, basic operations are available in Python. These are (with their respective commands): addition +, subtraction -, division /, multiplication *, exponentiation **, and the modulo operation (or remainder) %, and equality ==. Let's try them all.

Addition

```
print(2 + 2)
```

Subtraction

```
print(2 - 2)
```

```
### Division
```

```
print(2/2)
```

```
### Multiplication
```

```
print(2*2)
```

```
### Exponentiation
```

```
print(2**3)
```

```
### Remainder (modulor)
```

```
print(1%3)
```

```
### Equality
```

```
print(2 == 3)
```

```
print(2 == 2)
```

```
### ----- REASSIGNMENT OF VARIABLES -----
```

```
# Here we demonstrate the idea of reassignment. That is, assigning a value to an existing variable. This may seem trivial, but it will prove to be an essential tool later.
```

```
#
```

```
someinteger = someinteger + 4
```

```
print(someinteger)
```

```
### ----- FLOATS AND INTEGERS -----
```

```
# Here we provide a few quick comments on differences between floats and integers.
```

```
# As mentioned before, integers are the counting numbers, can be positive and negative, and never have either decimals or commas.
```

```
# Floats on the other hand are expressed with decimals, or in scientific notation. "Float" is short for floating point number, where the phrase "floating point" expresses the fact that some truncation error will exist depending on the bit-type of the float. A thorough discussion of bit-types and "machine epsilon" (a fundamental quantity for numerical computing) is beyond the scope of this workshop. In short, we'll say two things. (1) Even for computers, the number line is discrete and (2) At some point rounding will occur. We can do two quick calculations to demonstrate these ideas.
```

```
# For point (1) tell Python to print the operations  $1 + 1e-15$  and  $1 + 1e-16$ . What do you expect the answers to be? What are they really?
```

```
print(1 + 1e-15)
```

```
print(1 + 1e-16)
```

```
### For point (2) tell Python to print the operations  $1.200000 \cdot 10^{**8} + 2.5000 \cdot 10^{**(-7)}$  and  $1.200000 \cdot 10^{**8} + 2.5000 \cdot 10^{**(-8)}$  and  $1.200000 \cdot 10^{**8} + 2.5000 \cdot 10^{**(-9)}$ . What do you expect the answers to be? What are they really?
```

```
print(1.2*10**8 + 2.5*10**(-7))
```

```
print(1.2*10**8 + 2.5*10**(-8))
```

```
print(1.2*10**8 + 2.5*10**(-9))
```

```
### ----- WORKING WITH NUMBERS, TYPES, AND COERCION -----
```

Here we introduce a few functions that can be useful when working with collections of numbers.

You may be interested in the maximum, minimum, and absolute values of numbers. These functions exist in Python, and have sensible names. For example, consider the collection of numbers 5, 7, 3, 5, 8, 2. What would you guess is the name of the Python function to find the maximum, or the minimum?

Maximum

```
max(5, 7, 3, 5, 8, 2)
```

Minimum

```
min(5, 7, 3, 5, 8, 2)
```

Absolute value: How about the absolute value of a negative number, like -10?

```
abs(-10)
```

Types: We can also ask Python to tell us the type of number we are working with.

Integer type

```
type(10)
```

Float type

```
type(10.4)
```

String type

```
type("abcd")
```

Coercion: There can be instances where we need to "coerce" a type from what Python might first assume to what we actually want. An example of this is that Python assumes the number 10 is an integer. But maybe we need to treat 10 as a string (it could be part of a filename, directory, date, etc.). To coerce a number to a string, we use the `str()` function.

Try coercing the integer 10 into a string using the str() function. How can you tell it worked?

```
str(10)
```

On the other hand, we may need to coerce a string into a float. Try coercing the string '10' into a float using the float() function. How can you tell it worked?

```
float("10")
```

What about strings to integers?

```
int("10")
```

Floats to integers?

```
int(10.0)
```

Letters and punctuation to floats or integers?

```
float("a")
```

----- FUNCTIONS -----

Like the functions that are inherent in Python, we can write our own functions to use multiple times for variable inputs. The typical form of a function in python goes as follows.

```
# def function_name(parameter1, parameter2):
```

```
#     enter
```

```
#     here
```

```
#     any
```

```
#     code
```



```
# needed
# and comments
# return result_of_function

# For our first function, we will write a function that takes a parameter, and returns three times it
the value of the parameter.
```

```
def myFirstFunction(myParameter):
```

```
    # This is the inside of our first function. Everything we type here is only executed when the
function is called.
```

```
    # Lets add a print statement to the function for fun.
```

```
    print("Running my first function")
```

```
    return myParameter * 3
```

```
### Now let's run our function with different inputs.
```

```
myFirstFunction(11)
```

```
### If we assign an instance of running the function to a variable name, then the output from
the function is suppressed, and assigned to that variable name.
```

```
myNumber = myFirstFunction(111)
```

```
### ----- WORKING WITH TEXT -----
```

```
# Here we will begin learning some more Python and programming principles, but strictly in the
context of sequencing analysis. This type of data may not be your thing, but we invite you to
come along for this part of the workshop anyways as we will still learn vital programming
concepts. However, for examples of analyzing other types of data (e.g. spreadsheets of data,
images) we encourage you to consider taking any of the further Collaboratorys workshops
focusing on Python applications (Advanced Python, Machine Learning with Python, and Digital
Image Processing with Python).
```

```
### ----- QUOTATIONS MARKS -----
```

```
# First off, when defining a segment of text, or a string, how many quotes do we need to use?
```

```
# Try defining a string with single quotes.
```

```
teststring = 'stringcheese'
```

```
### Try defining a string with double quotes.
```

```
secondstring = "firststring"
```

```
### What if we mix single and double quotes?
```

```
jvteam = 'how many quotes?'
```

```
### What if our string is possessive (it contains an apostrophe)?
```

```
ownedstring = 'alex's string'
```

```
# Try using double quotes, or placing a slash in front of the apostrophe
```

```
### ----- LOGICAL IN/EXCLUSION, CONCATENATION, MULTIPLICATION -----
```

```
# in: We can use the "in" function and the "not in" function to search for the presence of one string (or motif) in another. Try asking Python if the string "ATA" is contained in the larger string "TATATATA"
```

```
"ATA" in "TATATATA"
```

```
### not in: Try asking Python if the string "banana" is not contained in the larger string "fruit bowl"
```

```
"banana" not in "fruit bowl"
```

+: The addition operation can be used to merge two strings together. This is called concatenation. Try concatenating the strings "AC" and "TG"

```
"AC" + "TG"
```

*: The multiplication symbol can be used to repeat strings. Try repeating the string "AT" 5 times.

```
"AT"*5
```

```
### ----- BASIC FUNCTIONS ON STRINGS -----
```

Python has several basic functions that can be used to easily access information about your string. To see a list of these, use the "dir()" function with the name of your string as the argument.

```
dir(guitarstring)
```

To use these functions on your string, type the name of your string, followed by a period, followed by the name of the function, followed by open and close parentheses.

upper: We can set all of the alphabet values in the string to be upper case values.

```
guitarstring.upper()
```

lower: We can set all of the alphabet values in the string to be lower case values.

```
guitarstring.upper().lower()
```

`### count:` We can ask Python for a count of all instances of a particular element in a string. With this function, we must place the element of interest within the parentheses. Try asking Python for all instances of the element 'e' in your string.

```
guitarstring.count('e')
```

`### find:` We can ask Python for the first index corresponding to the location of particular elements in a string. Again, with this function we must place the element of interest within the parentheses. Try asking Python for the location of the first instances of the element 'e' in your string.

```
guitarstring.find('e')
```

`### startswith:` We can ask Python if a particular string is the starting string in our given string. Again, with this function we must place the string of interest within the parentheses.

```
guitarstring.startswith('s')
```

`### len:` We can ask Python for the length of a string using the function `len()`. Unlike our previous functions, this function is used by placing the string (or its variable name inside of the parentheses). Try asking Python for the length of your string.

```
len(guitarstring)
```

`###` Now let's make a more useful function. This new function will calculate the GC content of a string sequence. To write this function, we will first practice with writing 'pseudocode'.

Pseudocode is the name given to the rough draft of the code that we write. This can be done on paper, in a separate text file, on a dry erase board. It is helpful to try and loosely structure the pseudocode as we would expect it to look when written.

`#` For this function, we can write our pseudocode in the following way.

```
# input is a sequence
#   count the occurrences of 'G' in the sequence, assign to a temporary variable
#   count the occurrences of 'C' in the sequence, assign to a temporary variable
#   add the number of 'G' and 'C' occurrences, assign to a temporary variable
#   divide the sum by the total sequence length, assign to a temporary variable
#   return the result
```

```
def gc_content(seq):
```

```
    # Here, our parameter name is seq, and it does not matter if seq has already been defined in
    the 'environment' outside of the function. This instance of seq is particular only to what happens
    inside of the gc_content function.
```

```
    gCount = seq.count('G') # Calculate number of 'G' in seq
    cCount = seq.count('C') # Calculate number of 'C' in seq
    gcCount = gCount + cCount # add numbers together
    totalCount = len(seq) # calculate length of seq
    gcContent = gcCount/totalCount # determine ratio of total to length
    return gcContent # return ratio
```

```
### Now let's use our function.
```

```
gc_content('ATCG')
```

```
### ----- IF-ELSE STATEMENTS -----
```

```
# The real utility of conditional statements is when using them with "if-else" statements. These
are statements that tell Python what to do "if" a given condition is True. "else", if the condition is
False, then Python does something else. These are extremely valuable statements to know
how to write. Let's try writing an if else statement that prints a string depending on the whether
one number is greater than another number. First, let's define a variable to represent our first
number.
```

```
mynumber = 10
```

```
### Now we'll write the if else statement.
```

```
if mynumber >= 4:  
    print('mynumber is greater than 4')  
else:  
    print('mynumber is less than 4')
```

Note how Python automatically formatted the statement for us. This is done to streamline how much we have to type (colons to separate the 'if' and 'else' conditions from the actions, whereas other languages require numerous parentheses) as well as to make the code easier to read in terms of an executing hierarchy.

```
### ----- IF-ELIF-ELSE STATEMENTS -----
```

Python is unique from other interpreted languages in that it has a dedicated tertiary conditional command, the 'elif' command, which is short of 'else if'. You may find that you have multiple conditional scenarios for which you want Python to do different things (for example, think about how many possibilities we had in our Truth table, or maybe when comparing numbers we are interested in exact equality as well as greater than or lesser than.). For these scenarios, we use the 'elif' statement. Let's try using the elif statement to tell Python to check if one number is greater than, exactly equal, or lesser than another number, and for each instance to print "greater than", "exactly equal" or "lesser than".

```
if mynumber > 10:  
    print('mynumber is greater than 10')  
elif mynumber == 10:
```

```
    print('mynumber is exactly equal to 10')
else:
    print('mynumber is less than 10')
```

```
### ----- Final function of the day -----
```

Instructions for writing this function. This new function should take a sequence as an input parameter (combinations of A, T, G, C). It will then search that sequence and, if the sequence starts with 'ATC', it should print the T content of the sequence (as we did with the GC content earlier). If the sequence starts with 'AGC', it will print 'Starting with AGC', and the number of times 'G' appears, but only if it does not follow 'A'. If the sequence starts with neither 'ATC' or 'AGC', then it will print 'Starting with neither ATC or AGC'.

```
def startswithATC(seq):
    if seq.startswith("ATC"):
        print(seq.count('T')/len(seq))
    elif seq.startswith("AGC"):
        gcount = seq.count("G") - seq.count("AG")
        print('Starting with AGC', gcount)
    else:
        print("Starting with neither ATC or AGC")
```